简介

这篇文章主要记录我对“Large steps in cloth simulation”中公式的理解及推导工程，以及如何用opengl实现这些公式。个人能力有限，没能渲染出很好的效果，但仍然认为这些理解这些公式很重要。

Large steps in cloth simulation 是布料模拟中一篇很经典的论文。这篇1998的论文目前为止被引用超过1700，并且我在网上搜索信息时，仍能发现国外很多大学将复现这篇论文作为课程大作业。

在弹簧质点系统，通常会使用三种约束力，即拉伸(stretch)力，剪切(shear)力，以及弯曲(bend)力，这三种约束由质点之间的距离计算出来。而在这篇论文中，也使用了这三种约束力，并且由三角形面片的形变来计算出来。

此外，这篇论文还使用了隐式求解器，即共轭梯度法。隐式方法需要解线性方程组，结果更加稳定，把时间步长，或者力设得很大也能正确求解。这就是能够"Large steps"的原因。

形变梯度(deformation gradient)

弹性物体最大的特点就是，在一定程度的变形后，还能缓慢恢复成最初始的形状。但是我们怎么衡量物体变形的程度，然后计算出每个节点的速度呢？最简单的就是使用形变梯度。

$$\mathbf{F} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}}$$

等式右边的分子就是物体现在位置的梯度，分母则是初始位置的梯度。在继续往下阅读之前，请先看看这篇介绍形变梯度的非常好的文章http://www.continuummechanics.org/deformationgradient.html#:~:text=The%20deformation%20gradient%20is%20used,are%20the%20source%20of%20stresses.&text=As%20is%20the%20convention%20in,defines%20the%20deformed%20current%20configuration.

我们可以编程来计算形变梯度

```
import numpy as np

# 形变前，物体（三角形）三个顶点的位置
pos0 = np.array([0,0])
pos1 = np.array([1,0])
pos2 = np.array([0,1])
e10 = pos1 - pos0
e20 = pos2 - pos0
```

```
d_X = np.array([e10,e20]) # 公式1的分母
minv = np.linalg.inv(d_X.T) #

# 形变后，物体的顶点位置
pos0_new = np.array([0,0])
pos1_new = np.array([1,1])
pos2_new = np.array([0,2])
e10_new = pos1_new - pos0_new
e20_new = pos2_new - pos0_new
d_x = np.array([e10_new,e20_new]) # 公式1的分子

F = np.dot(minv,d_x.T) # deformation Gradient
```

不过在“Large steps in cloth simulation ”(以下称论文[1])中，衡量物体物体形变程度并非是deformation gradient，而是一个很相似的东西，w

$$\mathbf{w} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}}$$

主要是将初始位置的梯度减少到了二维，并改了个名字叫uv。

$$\begin{bmatrix} \mathbf{w}_u & \mathbf{w}_v \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{x}_1 & \Delta\mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \Delta u_1 & \Delta u_2 \\ \Delta v_1 & \Delta v_2 \end{bmatrix}^{-1}$$

不知道大家看懂这个公式没有？至少我第一次看的时候就没看懂，我那时候在想，如果有个能够手算的例子就好了，至少能验证我脑海中那个设想是否正确。因此我会尽量多写一些简单的能够手算的例子。

首先假设一个三角形，最开始三个顶点的坐标分部是[0,0,0],[1,0,0],[0,1,0]。最开始这个三角形不应该有任何形变，所以它的三个的uv坐标是[0,0],[1,0],[0,1]。这样就能算出公式3中右边的一项了

$$\begin{bmatrix} \Delta u_1 & \Delta u_2 \\ \Delta v_1 & \Delta v_2 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

经过一段时间后，三角形的三个顶点的坐标仍然是[0,0,0],[1,0,0],[0,1,0]，那么计算现在顶点梯度

$$\Delta\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \Delta\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

稍微吐槽一下顶点梯度的下标。我发现有的主要有三种写法

```
//第一种
dx1 = p1 - p0
dx2 = p2 - p0
//第二种
dx10 = p1 - p0
dx20 = p2 - p0
//第三种
dx10 = p0 - p1
dx20 = p0 - p2
```

虽然写哪种凭个人喜好，但我还是倾向于第二种，因为能一眼看出来是哪个顶点和哪个顶点的梯度。最后w矩阵的结果为

$$[\mathbf{w}_u \quad \mathbf{w}_v] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

这玩意有什么用？可以算出为了把三角形掰回正常的不形变状态，而需要的力的大小和方法。如果你足够了解形变梯度，就会发现形变梯度和w矩阵有多么相似了。

不过在此之前，我们还需要定义“正常状态”是什么，参考[2]对于此处的讲解，也就是为什么要选形变梯度，为什么线弹性和neohookean模型各有优点等说得非常好。

就像之前说的，一个三角形最开始的三个顶点坐标分别为[0,0,0],[1,0,0],[0,1,0]，并且一开始处于未形变状态。

如果这个三角形现在的三个顶点坐标分别为[0,0,0],[1,0,0],[0,1,0]，那么它形变了吗？需要添加额外的力把它拉回初始状态吗？显然不需要，因为三个顶点就在原来的位置。

如果这个三角形现在的三个顶点坐标分别为[1,1,0],[2,1,0],[1,2,0]，那么它仅仅是位移了，没形变。同样不应该添加任何力。

如果这个三角形现在的三个顶点坐标分别为[0,0,0],[0,1,0],[-1,0,0]，那么形变了吗？它确实不在原来的位置，但它的面积/体积并没有变化，仅仅是旋转了90度，所以也不应该添加额外的力。

Stretch

如果这个三角形现在的三个顶点坐标分别为[0,0,0],[2,0,0],[0,1,0]，那么它确实形变了，面积变化了，我们的模型应该产生相应的力，把它拉回最初的形状。这里仅仅是第2个顶点拉伸了，我们应该添加一个拉伸力，将第2个顶点拉回来，不过拉回来的力度应该多大？

论文中的方法很简单，之间w矩阵两列取长度，再减去一个常量，再乘上一个常量即可。这和弹簧质点中的方法是一样的。算出的结果叫做Condition。对于拉伸力来说，Condition的结果是2x1矩阵，一个是u方向上的，一个v方向上的

$$\mathbf{C}_{stretch}(\mathbf{x}) = \alpha \begin{cases} ||\mathbf{w}_u(\mathbf{x})|| - \mathbf{b}_u = \sqrt{2^2 + 0^2 + 0^2} - 1 = 1 \\ ||\mathbf{w}_v(\mathbf{x})|| - \mathbf{b}_v = \sqrt{1^2 + 0^2 + 0^2} - 1 = 0 \end{cases}$$

写出来也很简单

```
float wuNorm = wu.norm();
float wvNorm = wv.norm();

float cu = alpha_stretch * (wuNorm - 1);
float cv = alpha_stretch * (wvNorm - 1);
```

算出Condition后，再算势能，或者叫能量，是个标量

$$E_{\mathbf{C}}(\mathbf{x}) = \frac{k}{2}\mathbf{C}(\mathbf{x})^T\mathbf{C}(\mathbf{x})$$

然而仅算出能量没有什么用，还让势能对顶点求导，得到每个顶点上的力

$$\mathbf{f}_i = \frac{-\partial E_{\mathbf{C}}}{\partial \mathbf{x}_i} = -k\frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i}\mathbf{C}(\mathbf{x})$$

对于每个顶点来说，力是一个3x1的矩阵。注意是对顶点位置求导哦。怎么对顶点位置求导呢？比如对于

$$\Delta\mathbf{x}_1 = \mathbf{x}_1 - \mathbf{x}_0$$

那么它对两个顶点求导分部为

$$\frac{\partial \Delta\mathbf{x}_1}{\partial \mathbf{x}_1} = \mathbf{I} \qquad \frac{\partial \Delta\mathbf{x}_1}{\partial \mathbf{x}_0} = -\mathbf{I}$$

I是单位矩阵。condition由w构成，而w由顶点位置x构成，所以求导很简单。在一个非常棒的开源库vegafem中的clothBW.cpp中，它是这么计算的

```
ClothBW::WuvInfo ClothBW::ComputeWuvInfo(const double
triangleUV[6])
{
  // distance of neighboring vertices in planar coordinates.
  // (delta_u1, delta_v1): planar vector from A to B,
  // (delta_u2, delta_v2): planar vector from B to A.
  double du1, du2, dv1, dv2;
```

```
  du1 = triangleUV[2] - triangleUV[0];
  dv1 = triangleUV[3] - triangleUV[1];
  du2 = triangleUV[4] - triangleUV[0];
  dv2 = triangleUV[5] - triangleUV[1];
  double Delta = 1.0/(du1*dv2-du2*dv1);
  WuvInfo info; // compute derivatives of wu and wv with respect to
vtx position x
  // 3x1 vector: wu = ( (x1-x0) dv2 - (x2-x0) dv1 ) / (du1 dv2 -
du2 dv1), xi is a 3x1 vector for vtx i on a triangle<x0,x1,x2>
  // 3x1 vector: wv = (-(x1-x0) du2 + (x2-x0) du1 ) / (du1 dv2 -
du2 dv1)

  info.pwupx[0] = (dv1 - dv2) * Delta;
  info.pwupx[1] = dv2 * Delta;
  info.pwupx[2] = -dv1 * Delta;
  info.pwvpx[0] = (du2 - du1) * Delta;
  info.pwvpx[1] = -du2 * Delta;
  info.pwvpx[2] = du1 * Delta;
  return info;
}
```

vegaFem是一个开源的有限元库，里面集成了很多论文的算法，在4.0版本中也包括参考[1]，下载地址为http://barbic.usc.edu/vega/download.html。github上的哪个vegafem版本为2.0。当然，在笔者本人的代码中是这么写的，用了Eigen库。

```
duv10 = uv1 - uv0;
duv20 = uv2 - uv0;
float det = 1 / (duv10(0) * duv20(1) - duv10(1) * duv20(0));

element_dwu[cnt * 2](0) = (duv10(1) - duv20(1)) * det;
element_dwu[cnt * 2](1) = duv20(1) * det;
element_dwu[cnt * 2](2) = -duv10(1) * det;

element_dwv[cnt * 2](0) = (duv20(0) - duv10(0)) * det;
element_dwv[cnt * 2](1) = -duv20(0) * det;
element_dwv[cnt * 2](2) = duv10(0) * det;
```

计算出dwu和dwv，之后就可以算力。在vegafem4.0中的clothBW.cpp中是这么算的

```
    double length_wu = len(wu);
    double length_wv = len(wv);
```

```
    Vec3d wun = wu / length_wu; //wu normalized
    Vec3d wvn = wv / length_wv; // wv normalized

    double alpha = alphas[tri];

    // --- compute stretch and shear energy ---
    // stretch energy Es = 0.5 Cs^T Cs
    // Cs = [Csu Csv]^T = alpha [ |wu|-bu |wv| -bv ]^T
    double Cstru = alpha * (length_wu - bu); // stretch energy in
u: Csu
    double Cstrv = alpha * (length_wv - bv); // stretch energy in
v: Csv

      *energy += 0.5 * materialGroups[group].tensileStiffness *
(Cstru*Cstru + Cstrv * Cstrv);
```

在另一个不错的开源库，地址为https://github.com/davvm/clothSim/blob/master/src/simLib/StretchCondition.cpp#L101，力是这么算的。对照公式看很快就能看明白。注意对每个顶点，力是一个3x1矩阵。

```
C0 = a * (wuNorm - bu);
C1 = a * (wvNorm - bv);

Real wuNorm = wu.norm();
dC0dP0 = a * dwudP0 * wu / wuNorm;
dC0dP1 = a * dwudP1 * wu / wuNorm;
dC0dP2 = a * dwudP2 * wu / wuNorm;

Real wvNorm = wv.norm();
dC1dP0 = a * dwvdP0 * wv / wvNorm;
dC1dP1 = a * dwvdP1 * wv / wvNorm;
dC1dP2 = a * dwvdP2 * wv / wvNorm;

forces.segment<3>(3 * m_inds[0]) -= k * (q.C0 * q.dC0dP0 + q.C1 *
q.dC1dP0);
forces.segment<3>(3 * m_inds[1]) -= k * (q.C0 * q.dC0dP1 + q.C1 *
q.dC1dP1);
forces.segment<3>(3 * m_inds[2]) -= k * (q.C0 * q.dC0dP2 + q.C1 *
q.dC1dP2);
```

然而只算出力，只能算显式步骤，我们之后还要让力对顶点位置和顶点速度求导，才能算出隐式步骤。不过这个之后再说。

怎么样，和弹簧质点中的stretch中很不一样吧？如果你还不理解为什么可以由形变梯度算出能量，能量又是怎么和力扯上关系的，那么你需要多阅读一些有限元的书籍和代码，比如尽量把这个维基里的每个词都弄懂https://en.wikipedia.org/wiki/Finite_strain_theory，我弄懂的方法主要是看代码然后纸笔推演了。

Shear

一个三角形最开始的三个顶点坐标分别为[0,0,0],[1,0,0],[0,1,0]，如果这个三角形现在的三个顶点坐标分别为[0,0,0],[1,0.5,0],[0.5,1,0]，那么W矩阵当然就变成了下面这样。

$$\begin{bmatrix} \mathbf{w}_u & \mathbf{w}_v \end{bmatrix} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \\ 0 & 0 \end{bmatrix}$$

由于三角形边产生了纯剪切形变，所以剪切力的Condition不再为零

$$\mathbf{C}_{shear}(\mathbf{x}) = \mathbf{w}_u^T \mathbf{w}_v = 1$$

注意我们说的纯剪切，也就是不带旋转的剪切。因为旋转并没改变面积/体积，算不上形变。

旋转算不上形变，所以需要用各种方法去除旋转分量，比如QR分解，SVD分解，顺带一提，taichi库中mpm99.py中的svd也是这个作用哦。以下代码分别来自第38行和51行。U,V都是旋转分量。？？？有疑问

```
U, sig, V = ti.svd(F[p])
stress = 2 * mu * (F[p] - U @ V.transpose()) @ F[p].transpose() +
ti.Matrix.identity(float, 2) * la * J * (J - 1)
```

参考[2]也讲述了如何用各种方法去除旋转分量。同样，由剪切导致的力可计算如下

```
float Shear = alpha_shear * wu.dot(wv);

Vector3f dcdx0 = alpha_shear * (dwu(0) * wv + dwv(0) * wu);
Vector3f dcdx1 = alpha_shear * (dwu(1) * wv + dwv(1) * wu);
Vector3f dcdx2 = alpha_shear * (dwu(2) * wv + dwv(2) * wu);

forces.segment<3>(idx0 * 3) += -k_shear * Shear * dcdx0;
forces.segment<3>(idx1 * 3) += -k_shear * Shear * dcdx1;
forces.segment<3>(idx2 * 3) += -k_shear * Shear * dcdx2;
```

我们前两个力并不复杂，因为第三个bend力推导能推几页纸qwq。所以bend才是我们的大boss。

为了快速获取正反馈，我们先用显式方法，把两个力加上去。到这里为止的代码文件为step01。不过你想把这个工程运行起来也是需要一番魄力的，因为我用了freeglut,glew库，以及数学运算库eigen。【所以像taichi这样可以一键运行的真是yyds】

阻尼

阻尼在运动方程中非常常见，通常和速度有关，速度越大阻尼越大，以其将速度减少。如果不加阻尼，现在你会看到物体一直在来回移动，丝毫没有停下来的意思。阻尼也是一种力，所以可以直接将结果加到力上。阻尼计算公式如下

$$\mathbf{d} = -k_d \frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}} \dot{\mathbf{C}}(\mathbf{x})$$

上面等式右边最右边那项是Condition对时间求导，也就是

$$\dot{\mathbf{C}}(\mathbf{x}) = \frac{\partial \mathbf{C}(\mathbf{x})}{\partial t} = \frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{t}} = \frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}} \mathbf{v}$$

所以直接让Condition对顶点求导再乘上顶点的速度就形了。最后的结果是个标量。

对于Stretch来说，阻尼导致的力计算如下

```
float dcudt = dcudx0.dot(v0) + dcudx1.dot(v1) + dcudx2.dot(v2);
float dcvdt = dcvdx0.dot(v0) + dcvdx1.dot(v1) + dcvdx2.dot(v2);

forces.segment<3>(idx0 * 3) += -damping_stretch * (dcudt * dcudx0 +
dcvdt * dcvdx0);
forces.segment<3>(idx1 * 3) += -damping_stretch * (dcudt * dcudx1 +
dcvdt * dcvdx1);
forces.segment<3>(idx2 * 3) += -damping_stretch * (dcudt * dcudx2 +
dcvdt * dcvdx2);
```

对于Shear来说，计算如下

```
float dcdt = dcdx0.dot(v0) + dcdx1.dot(v1) + dcdx2.dot(v2);

forces.segment<3>(idx0 * 3) += -damping_shear * dcdt * dcdx0;
forces.segment<3>(idx1 * 3) += -damping_shear * dcdt * dcdx1;
forces.segment<3>(idx2 * 3) += -damping_shear * dcdt * dcdx2;
```

至于Bend，它将会作为大Boss出场，请做好准备。

至此的代码文件为Step02。

隐式步骤

广义动力学公式如下

$$\frac{d}{dt}\begin{bmatrix}\mathbf{x}\\ \mathbf{v}\end{bmatrix}=\frac{d}{dt}\begin{bmatrix}\mathbf{v}\\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x},\mathbf{v})\end{bmatrix}$$

其中x是位置，v是速度，M是结点的质量，是标量。f是结点的力。经过一顿操作，它变成了

$$(\mathbf{I}-h\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{v}}-h^2\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{x}})\Delta v=h\mathbf{M}^{-1}(\mathbf{f}_0+h\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{v}_0)$$

其中I是单位矩阵，h是步长，是标量，需要足够小。dfdv和dfdv都是矩阵，行列数都是node_num*3,node_num是结点数量，乘以3是因为坐标轴有三个方向。f0和v0分别是结点的力和速度，都是(node_num乘3)行一列的矩阵。

这样上面的公式就变成了一个线性方程组

$$lhs\Delta\mathbf{v}=rhs$$

其中lhs是一个行列均为node_num乘3的矩阵，rhs是一个(node_num乘3)行一列的矩阵。求解出来的速度的变化量，也是(node_num乘3)行一列的矩阵。不过上面的公式写出来就像上面这样子

```
lhs = identity - h / mass * dfdv - h * h / mass * dfdx;
rhs = h / mass * (forces + h * dfdx * v);
dv = lhs.inverse() * rhs;
```

如果想简单一些，直接对lhs求逆再乘rhs就能结果。不过在这里，重要的是怎么构建dfdx和dfdv。首先是力对顶点求导

$$\mathbf{K}_{ij}=\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_i}=-k(\frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i}\frac{\partial \mathbf{C}(\mathbf{x})^T}{\partial \mathbf{x}_j}+\frac{\partial^2 \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i\partial \mathbf{x}_j}\mathbf{C}(\mathbf{x}))$$

力对顶点的求导的结果是3x3矩阵。不过在此之前，Condition还需要对顶点位置求两次导，结果也是3x3矩阵。stretch部分的代码如下

```
d2C0dP0dP0 = (a / wuNorm) * dwudP0 * dwudP0 * wuMatrix;
d2C0dP0dP1 = (a / wuNorm) * dwudP0 * dwudP1 * wuMatrix;
d2C0dP0dP2 = (a / wuNorm) * dwudP0 * dwudP2 * wuMatrix;

Matrix3 df0dP0 = -k * (q.dC0dP0 * q.dC0dP0.transpose() + q.C0 *
q.d2C0dP0dP0 + q.dC1dP0 * q.dC1dP0.transpose() + q.C1 *
q.d2C1dP0dP0);
Matrix3 df0dP1 = -k * (q.dC0dP0 * q.dC0dP1.transpose() + q.C0 *
q.d2C0dP0dP1 + q.dC1dP0 * q.dC1dP1.transpose() + q.C1 *
q.d2C1dP0dP1);
Matrix3 df0dP2 = -k * (q.dC0dP0 * q.dC0dP2.transpose() + q.C0 *
q.d2C0dP0dP2 + q.dC1dP0 * q.dC1dP2.transpose() + q.C1 *
q.d2C1dP0dP2);
```

每个三角形有三个顶点，每个顶点都有自己的力，所以一个三角形应该求出来9个矩阵，每个矩阵都是3x3的。阻尼也需要对顶点位置求导，公式如下，对单个顶点来说，结果是3x3矩阵

$$\frac{\partial \mathbf{d}_i}{\partial \mathbf{x}_i} = -k(\frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i}\frac{\partial \dot{\mathbf{C}}(\mathbf{x})}{\partial \mathbf{x}_j}^T + \frac{\partial^2 \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j}\dot{\mathbf{C}}(\mathbf{x}))$$

与之前相比不过是有两项从Condition变成Condition对时间求导了，不过根据参考[4]的注释，如果按照上面的公式，等式右边就不对称了，待会儿就不能用共轭梯度了。所以所以将上面的公式修改如下

$$\frac{\partial \mathbf{d}_i}{\partial \mathbf{x}_i} \approx -k(\frac{\partial^2 \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j}\dot{\mathbf{C}}(\mathbf{x}))$$

省略的部分很小，所以实际并无影响。但省略之后我们就能愉快地使用共轭梯度了。
Stretch的部分的代码如下

```
df0dx0 = -damping_stretch * (d2cudx0x0 * dcudt + d2cvdx0x0 *
dcvdt);
df0dx1 = -damping_stretch * (d2cudx0x1 * dcudt + d2cvdx0x1 *
dcvdt);
df0dx2 = -damping_stretch * (d2cudx0x2 * dcudt + d2cvdx0x2 *
dcvdt);


df1dx0 = -damping_stretch * (d2cudx1x0 * dcudt + d2cvdx1x0 *
dcvdt);
df1dx1 = -damping_stretch * (d2cudx1x1 * dcudt + d2cvdx1x1 *
dcvdt);
df1dx2 = -damping_stretch * (d2cudx1x2 * dcudt + d2cvdx1x2 *
dcvdt);

df2dx0 = -damping_stretch * (d2cudx2x0 * dcudt + d2cvdx2x0 *
dcvdt);
df2dx1 = -damping_stretch * (d2cudx2x1 * dcudt + d2cvdx2x1 *
dcvdt);
df2dx2 = -damping_stretch * (d2cudx2x2 * dcudt + d2cvdx2x2 *
dcvdt);
```

这些临时矩阵算好了之后，应该放进一个超大矩阵dfdx中，这个矩阵有(node_num乘3)行，(node_num乘3)列，并且对称的。比如df0dx1，是顶点0上的力对顶点1的位置求导，所以应该将它放进从第0行第3列开始的3x3矩阵中，用Eigen可写成如下

```
dfdx.block<3, 3>(idx0 * 3, idx1 * 3) += df0dx1;
```

idx0就是这个三角形中第0个顶点的索引。如果使用Eigen提供的稀疏矩阵，就是如下形式

```
for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j)
dfdx.coeffRef(3 * idx[0] + i, 3 * idx[1] + j) += df0dx1(i, j);
```

类似这样组装矩阵在有限元中是常见操作。刚才我们都在求dfdx，而现在要求dfdv，也就是力对结点速度求导，公式如下

$$\frac{\partial \mathbf{d}_i}{\partial \mathbf{v}_j} = -k_d \frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}_i} \frac{\partial \dot{\mathbf{C}}(\mathbf{x})}{\partial \mathbf{v}_i}^T \qquad \frac{\partial \dot{\mathbf{C}}(\mathbf{x})}{\partial \mathbf{v}_i} = \frac{\partial}{\partial \mathbf{v}} (\frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}}^T \mathbf{v}) = \frac{\partial \mathbf{C}(\mathbf{x})}{\partial \mathbf{x}}$$

stretch部分代码如下。这点对上面的dfdx也是一样的。

```
df0dv0 = -damping_stretch * (dcudx0 * dcudx0.transpose() + dcvdx0 *
dcvdx0.transpose());
df0dv1 = -damping_stretch * (dcudx0 * dcudx1.transpose() + dcvdx0 *
dcvdx1.transpose());
df0dv2 = -damping_stretch * (dcudx0 * dcudx2.transpose() + dcvdx0 *
dcvdx2.transpose());

df1dv0 = -damping_stretch * (dcudx1 * dcudx0.transpose() + dcvdx1 *
dcvdx0.transpose());
df1dv1 = -damping_stretch * (dcudx1 * dcudx1.transpose() + dcvdx1 *
dcvdx1.transpose());
df1dv2 = -damping_stretch * (dcudx1 * dcudx2.transpose() + dcvdx1 *
dcvdx2.transpose());

df2dv0 = -damping_stretch * (dcudx2 * dcudx0.transpose() + dcvdx2 *
dcvdx0.transpose());
df2dv1 = -damping_stretch * (dcudx2 * dcudx1.transpose() + dcvdx2 *
dcvdx1.transpose());
df2dv2 = -damping_stretch * (dcudx2 * dcudx2.transpose() + dcvdx2 *
dcvdx2.transpose());
```

dfdx和dfdv都算出来了，接下来解线性方程组就像了。可以简单求逆，但是lhs也就是K矩阵是个稀疏矩阵，求逆显然不划算。因此我们可以使用共轭梯度。

之所以说K是稀疏矩阵，是因为在我们的网格编排中，如果只考虑Stretch和Shear，那么一个结点最多之和除自己之外的6个结点有关，例如在下面的三角形网格中，只和左上，右下，上，下，左，右这6个结点有关系，因为只有这些结点与中心点共享了一个三角形。这意味在K每行的node_num乘3列中，最多只有7乘3列有数字，其它列全是零。

至于K是对称，当然是因为力的相互作用原理了。

K是稀疏矩阵，所以我们显然不可能以稠密矩阵的方式存它，不然太浪费空间了。

可以直接使用Eigen库的库方法存稀疏矩阵，之后求解也方便。

自己写一个标准的稀疏矩阵处理程序，可参考bridson的

笔者自己用的是另一种，既然K最多只有21列，并且编号和相对位置是固定的，那我直接把这21列存下了。比较方便。这种方法在taichi库的cg_poisson.py也能见到。

共轭梯度

共轭梯度请看这篇。这篇论文用了Modified Precondition Conjugate Gradient，笔者只实现了Conjugate Gradient。

至此的代码的文件可见

Bend

很高兴你读到了这里，接下来我们只要把bend力算出来，整个物理公式计算部分就完成了。必须隆重介绍了参考[5]，里面很详细地介绍了Bend的各项是怎么推导出来。

首先，我们希望相邻的三角形是平的，不弯曲，也就是它们的法向量相同，二面角为零。法向量不同的程度用二面角来衡量，二面角越大，就说明模型应该产生更大的内力，来把两个三角形的法向量弄相同。

Bend的Condition如下

$$\mathbf{C}(\mathbf{x}) = \theta$$

有两个很重要的关系是

$$\sin\theta = (\mathbf{n}_0 \times \mathbf{n}_1) \cdot \mathbf{e}_{12} \qquad \cos\theta = \mathbf{n}_0 \cdot \mathbf{n}_1$$

n0和n1分别是两个三角形的法向量。e12是三角形共享边的单位向量。另一个很重要的公式是三角形余弦公式

$$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}||||\mathbf{v}|| \cos\alpha_{uv}$$

接下来这个公式会悄无声息地出现，并且有时候u是单位向量，那么等式会直接少掉一项。

$$\hat{\mathbf{u}} \cdot \mathbf{v} = ||\mathbf{v}|| \cos\alpha_{uv}$$

所以有的公式推导着推导着突然少了一项，就是因为这个原因。

接下来，我们的目标是，算角度对顶点位置的一次导，角度对顶点位置的二次导，角度对顶点速度的一次导，最终目标是算出力对顶点位置求导，力对顶点速度求导，用于组装到刚度矩阵中去。如果你丝毫不知道从哪里开始算起的话，那么下面的内容就是为你准备的。

这部分的内容属于离散微分几何(discrete differentia geometry)，所以如果你去找《微分几何》或者《张量分析》是找不到任何有用的东西的。

碳基生物喜欢模块化的东西，所以接下来我们一步一步来拆解。首先，我们的两个顶点是这样的

```
2---3
| \ |
0---1
```

参考[5]的顶点编号，边的编号与这篇文章的不同。并且我认为它的下标并不是简洁易懂。

第一步：各个边的向量及其单位向量。

```
v10 = p1 - p0;
v20 = p2 - p0;
v23 = p2 - p3;
v13 = p1 - p3;
v12 = p1 - p2;

e10 = v10.normalized();
e20 = v20.normalized();
e23 = v23.normalized();
e13 = v13.normalized();
e12 = v12.normalized();
```

第二步：算余弦和法向量

也是一刀秒掉。

```
c00 = e10.dot(e20);//c00是指第0个三角中，顶点0的余弦
c01 = e10.dot(e12);
c02 = -e20.dot(e12);

c13 = e13.dot(e23);
c11 = e12.dot(e13);
c12 = -e23.dot(e12);

n0 = (e12.cross(e10)).normalized();
n1 = (e23.cross(e12)).normalized();
```

第三步：计算单位化的binormal

也就是和法向量以及切向量都正交的那个家伙。这里的binormal是指在一个三角形里，由三角形某个顶点指向对边的方向。代码可如下写

```
b00 = (e01 - e21 * (e21.dot(e01))).normalized();
b01 = (-e01 - e02 * (e02.dot(-e01))).normalized();
b02 = (-e02 - e01 * (e01.dot(-e02))).normalized();

b13 = (e32 - e21 * (e21.dot(e32))).normalized();
b12 = (-e32 - e31 * (e31.dot(-e32))).normalized();
b11 = (-e31 - e32 * (e32.dot(-e31))).normalized();
```

或者直接叉积也可以，注意方向。要算的结果是由三角形某个顶点指向对边的方向，因此法向量需要在边向量的左边。

```
b00 = -e12.cross(n0);
b01 = -e20.cross(n0);
b02 = e10.cross(n0);

b13 = e12.cross(n1);
b12 = -e13.cross(n1);
b11 = e23.cross(n1);
```

第四步：计算顶点到到对边的距离

结果为标量。原理为很简单的向量点积，binormal是单位向量，用字母m表示。字母m上面那个尖代表它是单位向量。

$$\hat{\mathbf{m}}_{00} \cdot \mathbf{v}_{10} = ||\mathbf{v}_{10}|| \cos \alpha_{00}$$

```
d00 = b00.dot(v10);//d00是指第0个三角形中，顶点0到对边的距离
d01 = b01.dot(-v12);
d02 = b02.dot(-v20);

d11 = b11.dot(-v13);
d12 = b12.dot(v12);
d13 = b13.dot(v13);
```

第五步：计算弯曲的Condition

也就是二面角的角度

$$\sin \theta = (\mathbf{n}_0 \times \mathbf{n}_1) \cdot \mathbf{e}_{12} \qquad \cos \theta = \mathbf{n}_0 \cdot \mathbf{n}_1$$

```
Real sinTheta = n1.dot(b00);
Real cosTheta = n0.dot(n1);
theta = atan2(sinTheta, cosTheta);
```

第六步：计算法向量对顶点的导数

法向量为单位向量。下面公式的计算结果是3x3的矩阵。

$$\nabla\hat{\mathbf{n}} = \nabla(\frac{\mathbf{n}}{||\mathbf{n}||}) = \mathbf{n}\nabla(\frac{1}{||\mathbf{n}||}) + \frac{1}{||\mathbf{n}||}\nabla\mathbf{n}$$

而且

$$\nabla(\frac{1}{||\mathbf{n}||}) = -\frac{1}{||\mathbf{n}||^2}\nabla(||\mathbf{n}||) = \frac{-1}{||\mathbf{n}||^2}(\hat{\mathbf{n}}^T\nabla\mathbf{n})$$

用上面两个式子化简可得

$$\nabla\hat{\mathbf{n}} = (1 - \hat{\mathbf{n}}\hat{\mathbf{n}}^T)\frac{\nabla\mathbf{n}}{||\mathbf{n}||}$$

这也是参考[5]第8页第一个公式。这就是将单位法向量求导转换为了对法向量求导。又因为

$$\nabla(\mathbf{n}) = \nabla(\mathbf{u}\times\mathbf{v}) = \frac{\mathbf{u}\times\nabla\mathbf{v} - \mathbf{v}\times\nabla\mathbf{u}}{||\mathbf{u}\times\mathbf{v}||}$$

参考[5]的第九页中间那个公式我看不懂，暂且跳过。总之经过一顿操作可以得到下面的式子

$$\nabla\hat{\mathbf{n}} = \frac{(\mathbf{u}\times\hat{\mathbf{n}})\hat{\mathbf{n}}^T\nabla\mathbf{v} - (\mathbf{v}\times\hat{\mathbf{n}})\hat{\mathbf{n}}^T\nabla\mathbf{u}}{||\mathbf{n}||}$$

对于第0个三角形的法向量，它的导数就应该是

$$\nabla\hat{\mathbf{n}}_0 = \frac{(\mathbf{e}_{12}\times\hat{\mathbf{n}}_0)\hat{\mathbf{n}}_0^T\nabla\mathbf{e}_{10} - (\mathbf{e}_{10}\times\hat{\mathbf{n}}_0)\hat{\mathbf{n}}_0^T\nabla\mathbf{e}_{12}}{||\mathbf{n}_0||}$$

首先是边的单位向量乘单位法向量再除以法向量长度，可以转换为单位binormal除以距离，如下

$$\frac{(\mathbf{e}_{12}\times\hat{\mathbf{n}}_0)}{||\mathbf{n}_0||} = \frac{||\mathbf{e}_0||(\hat{\mathbf{e}}_{12}\times\hat{\mathbf{n}}_0)}{||\mathbf{n}_0||} = -\frac{||\mathbf{e}_{12}||\hat{\mathbf{m}}_{00}}{||\mathbf{n}_0||} = -\frac{\hat{\mathbf{m}}_{00}}{h_{00}}$$

这也是参考[2]第17页第一个公式。那么对于第0个三角形的法向量，导数为

$$\nabla \hat{\mathbf{n}}_0 = -\frac{\hat{\mathbf{m}}_{00}}{h_{00}}\hat{\mathbf{n}}_0^T\nabla \mathbf{e}_{10} - \frac{\hat{\mathbf{m}}_{02}}{h_{01}}\hat{\mathbf{n}}_0^T\nabla \mathbf{e}_{12}$$

接下来让边的单位向量，对各个顶点求导。再复习一遍

$$\begin{aligned}
\nabla \mathbf{e}_{10} &= \mathbf{x}_1 - \mathbf{x}_0 = (-I, I, 0, 0) \\
\nabla \mathbf{e}_{20} &= \mathbf{x}_2 - \mathbf{x}_0 = (-I, 0, I, 0) \\
\nabla \mathbf{e}_{12} &= \mathbf{x}_1 - \mathbf{x}_2 = (0, I, -I, 0) \\
\nabla \mathbf{e}_{13} &= \mathbf{x}_1 - \mathbf{x}_3 = (0, I, 0, -I) \\
\nabla \mathbf{e}_{23} &= \mathbf{x}_1 - \mathbf{x}_3 = (0, 0, I, -I)
\end{aligned}$$

那么第0个法向量对第0个顶点求导，e10求导完是-1，e12求导完是0，结果是一个3x3矩阵。

$$\nabla_{\mathbf{x}_0} \hat{\mathbf{n}}_0 = \frac{\hat{\mathbf{m}}_{00}}{d_{00}}\hat{\mathbf{n}}_0^T$$

第0个法向量对第1个顶点求导，e10求导完是1，e12求导完是1，结果如下

$$\nabla_{\mathbf{x}_1} \hat{\mathbf{n}}_0 = -\frac{(\hat{\mathbf{m}}_{00} + \hat{\mathbf{m}}_{02})}{d_{01}}\hat{\mathbf{n}}_0^T = \frac{\hat{\mathbf{m}}_{01}}{d_{01}}\hat{\mathbf{n}}_0^T$$

对第二个顶点，e10求导完是0，e12求导完是-1，结果如下

$$\nabla_{\mathbf{x}_2} \hat{\mathbf{n}}_0 = \frac{\hat{\mathbf{m}}_{02}}{d_{02}}\hat{\mathbf{n}}_0^T$$

对于第三个顶点，e10和e12求导完结果都是0。第1个三角形的法向量求导同理。写成代码如下

```
dn0dx0 = b00 * n0.transpose() / d00;
dn0dx1 = b01 * n0.transpose() / d01;
dn0dx2 = b02 * n0.transpose() / d02;
dn0dx3 = Matrix3::Zero();

dn1dx0 = Matrix3::Zero();
dn1dx1 = b11 * n1.transpose() / d11;
dn1dx2 = b12 * n1.transpose() / d12;
dn1dx3 = b13 * n1.transpose() / d13;
```

第七步：计算角度对顶点的导数

也就是参考[5]第16页公式(18)~(21)。角度求导公式为

$$\nabla\theta = -\frac{\nabla(\hat{\mathbf{n}}_0 \cdot \hat{\mathbf{n}}_1)}{\sin\theta} = -\frac{\hat{\mathbf{n}}_0^T\nabla\hat{\mathbf{n}}_1 + \hat{\mathbf{n}}_1^T\nabla\hat{\mathbf{n}}_0}{\sin\theta}$$

法向量求导之前已经推导过了，但是现在我们需要将叉积换个位置，也就是使用公式

$$(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w} = (\mathbf{w}\times\mathbf{u})\cdot\mathbf{v}$$

那么就是下面这个

$$-(\hat{\mathbf{n}}_0\times\mathbf{e}_{12})\cdot\hat{\mathbf{n}}_1 = (\hat{\mathbf{n}}_0\times\hat{\mathbf{n}}_1)\cdot\mathbf{e}_{12} = (\hat{\mathbf{e}}_{12}\cdot\mathbf{e}_{12})\sin\theta = ||\mathbf{e}_{12}||\sin\theta$$

上面这个将sin消去的变换详细请看参考[5]第12~13页的公式。消去后得到下面的公式：

$$\frac{\hat{\mathbf{n}}_0^T\nabla\hat{\mathbf{n}}_1}{\sin\theta} = \frac{||\mathbf{e}_{12}||\hat{\mathbf{n}}_1^T\nabla\mathbf{e}_{32} - (\hat{\mathbf{e}}_{12}\cdot\mathbf{e}_{32})\hat{\mathbf{n}}_1^T\nabla\mathbf{e}_{12}}{||\mathbf{n}_1||}$$

以及

$$\frac{\hat{\mathbf{n}}_1^T\nabla\hat{\mathbf{n}}_0}{\sin\theta} = \frac{-||\mathbf{e}_{12}||\hat{\mathbf{n}}_0^T\nabla\mathbf{e}_{20} + (\hat{\mathbf{e}}_{12}\cdot\mathbf{e}_{20})\hat{\mathbf{n}}_0^T\nabla\mathbf{e}_{12}}{||\mathbf{n}_0||}$$

由因为

$$2A_0 = ||\mathbf{n}_0|| = ||\mathbf{e}_{12}||d_{00}$$

A0是三角形9的面积。再结合边的单位向量的求导公式。

$$\nabla_{\mathbf{x}_0}\theta = -\frac{1}{d_{00}}\hat{\mathbf{n}}_0^T \qquad \nabla_{\mathbf{x}_3}\theta = -\frac{1}{d_{13}}\hat{\mathbf{n}}_1^T$$

再结合参考[5]第14~16页的公式，就得到了下面的公式。这玩意推了5页多，实在太多了，所以细节还是请看参考[5]吧。

$$\nabla_{\mathbf{x}_1}\theta = \frac{\cos\alpha_{02}}{d_{01}}\hat{\mathbf{n}}_0^T + \frac{\cos\alpha_{12}}{d_{11}}\hat{\mathbf{n}}_1^T$$

$$\nabla_{\mathbf{x}_2}\theta = \frac{\cos\alpha_{01}}{d_{02}}\hat{\mathbf{n}}_0^T + \frac{\cos\alpha_{11}}{d_{12}}\hat{\mathbf{n}}_1^T$$

```
dThetadP0 = -n0 / d00;
dThetadP1 = c02 * n0 / d01 + c12 * n1 / d11;
dThetadP2 = c01 * n0 / d02 + c11 * n1 / d12;
dThetadP3 = -n1 / d13;
```

第八步：距离对顶点求导

由于我们算的binormal是顶点到对边的单位向量，所以如果顶点每向沿着法向量的方向移动一个量，必然导致这个点到对边的距离减少这个量点积刚才算出来的binormal

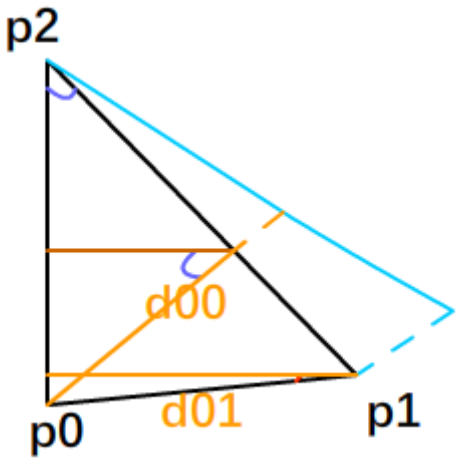
$$\nabla_{\mathbf{x}_0} d_{00} = -\hat{\mathbf{m}}_{00}^T$$

举个例子，一个二维三角形三个顶点分部是p0 = [0,0],p1 = [2,0],p2 = [0,2]。那点p0的binormal很明显就是a = [0.707,0.707]。如果这个点移动了b= [0.4,0]，那么这个点离对边的距离，减少了向量b的长度，也就是0.2828。但是巧了，a点积b结果的也是这个值。这种情况就是因为一开始提到的三角形余弦公式。

因此可以说某个点到对边的距离，对这个点求导，结果是这个点的binormal乘上-1。

那么就有人说，如果不移动p0，偏要移动p1，那么随着p1的移动，p0到对边的距离会如何变化？

假如p1移动了一个向量a，我们先让这个这个向量点积点p0的binormal，毕竟如果往别的方向移动的话，是不会增加d00的长度的。



现在已经求出了蓝色虚线的长度，也就是向量a点积binormal的结果，那么与蓝色虚线平行的橙色虚线的长度是多少？那么这样求相似三角形就可以了。很显然，蓝色虚线的长度比橙色虚线的长度，等于d01比图中红色实线的长度，而后者可通过d00乘以cos(角021)求出来，那么就得到了下面这个式子

$$\nabla_{\mathbf{x}_1} d_{00} = \frac{d_{00}}{d_{01}} \cos\alpha_{02} \hat{\mathbf{m}}_{00}^T$$

剩下移动p2导致的d00的改变也用相似的方法可求得。最后，无论我们如何移动p3，p0到对边的距离都不会有任何改变。所以对其求导结果是0。

$$\nabla_{\mathbf{x}_2} d_{00} = \frac{d_{00}}{d_{02}} \cos\alpha_{01} \hat{\mathbf{m}}_{00}^T \qquad \nabla_{\mathbf{x}_3} d_{00} = 0$$

这些公式就是参考[5]第21页前4个公式。代码可以这么写

```
dd00dP0 = -b00;
dd00dP1 = b00 * -v12.dot(v20) / v12.dot(v12);
dd00dP2 = b00 * v12.dot(v10) / v12.dot(v12);
dd00dP3 = Vector3::Zero();
```

这是因为

$$\frac{-\mathbf{v}_{12}\cdot\mathbf{v}_{20}}{||\mathbf{v}_{12}||^2}=\frac{\cos\alpha_{02}||\mathbf{v}_{12}||||\mathbf{v}_{20}||}{||\mathbf{v}_{12}||^2}=\frac{\cos\alpha_{02}||\mathbf{v}_{20}||}{||\mathbf{v}_{12}||}$$

再用一遍三角形余弦定理即可推出上式。

第九步：余弦对顶点求导

首先对于微分余弦

$$\nabla\cos\alpha_{01}=\nabla(\hat{\mathbf{e}}_{10}\cdot\hat{\mathbf{e}}_{12})=\hat{\mathbf{e}}_{10}\hat{\mathbf{e}}_{12}^T+\hat{\mathbf{e}}_{12}\hat{\mathbf{e}}_{10}^T=$$
$$\hat{\mathbf{e}}_{10}(1-\hat{\mathbf{e}}_{12}\hat{\mathbf{e}}_{12}^T)\frac{\nabla\mathbf{e}_{12}}{||\mathbf{e}_{12}||}+\hat{\mathbf{e}}_{12}(1-\hat{\mathbf{e}}_{10}\hat{\mathbf{e}}_{10}^T)\frac{\nabla\mathbf{e}_{10}}{||\mathbf{e}_{10}||}$$

还记得单位向量怎么求导吗？再推一遍

$$\mathbf{e}_{10}=\mathbf{x}_1-\mathbf{x}_0\qquad\frac{\partial\mathbf{e}_{10}}{\partial\mathbf{x}_1}=\mathbf{I}\qquad\frac{\partial\mathbf{e}_{10}}{\partial\mathbf{x}_0}=-\mathbf{I}$$
$$\mathbf{e}_{12}=\mathbf{x}_1-\mathbf{x}_2\qquad\frac{\partial\mathbf{e}_{12}}{\partial\mathbf{x}_1}=\mathbf{I}\qquad\frac{\partial\mathbf{e}_{12}}{\partial\mathbf{x}_2}=-\mathbf{I}$$

上面的e是3x1矩阵，I是3x3单位矩阵。那么余弦对各个点求导结果也很好写了

$$\nabla_{\mathbf{x}0}\cos\alpha_{01}=-\frac{\hat{\mathbf{e}}_{12}^T(1-\hat{\mathbf{e}}_{10}\hat{\mathbf{e}}_{10}^T)}{||\mathbf{e}_{10}||}$$

参考[5]第23页的推导我没看懂。希望有大佬能够解释。经过一番操作，代码可写为如下

```
    dc01dP0 = -b02 * b00.dot(v10) / v10.dot(v10);
    dc01dP2 = -b00 * b02.dot(v12) / v12.dot(v12);
    dc01dP1 = -dc01dP0 - dc01dP2;
    dc01dP3 = Vector3::Zero();

    dc02dP0 = -b01 * b00.dot(v20) / v02.dot(v20);
    dc02dP1 = b00 * b01.dot(v12) / v21.dot(v12);
    dc02dP2 = -dc02dP0 - dc02dP1;
```

```
    dc02dP3 = Vector3::Zero();

    dc11dP0 = Vector3::Zero();
    dc11dP2 = -b13 * b12.dot(v12) / v21.dot(v12);
    dc11dP3 = -b12 * b13.dot(v13) / v31.dot(v13);
    dc11dP1 = -dc11dP2 - dc11dP3;

    dc12dP0 = Vector3::Zero();
    dc12dP1 = b13 * b11.dot(v12) / v21.dot(v12);
    dc12dP3 = -b11 * b13.dot(v23) / v32.dot(v23);
    dc12dP2 = -dc12dP1 - dc12dP3;
```

第十步：求角度对顶点的二阶导

这一步倒很简单。角度对顶点的一次导结果如下：

```
dThetadP0 = -n0 / d00;
dThetadP1 = c02 * n0 / d01 + c12 * n1 / d11;
dThetadP2 = c01 * n0 / d02 + c11 * n1 / d12;
dThetadP3 = -n1 / d13;
```

这里面的每个因子都能对顶点再求一遍导，因此我们只需要把这些分部求导就行了，部分代码如下

```
d2ThetadP0dP0 = -dn0dP0 / d00 + n0 * dd00dP0.transpose() / (d00 *
d00);
d2ThetadP0dP1 = -dn0dP1 / d00 + n0 * dd00dP1.transpose() / (d00 *
d00);
d2ThetadP0dP2 = -dn0dP2 / d00 + n0 * dd00dP2.transpose() / (d00 *
d00);
d2ThetadP0dP3 = -dn0dP3 / d00 + n0 * dd00dP3.transpose() / (d00 *
d00);
```

至此，算dfdx和dfdv和force的组件都已经计算完毕，然后组装矩阵即可。

之后只需要亿点点优化和亿点点渲染技巧可以生成漂亮的布料的模拟了。不过我在此时已经打空了大半的血槽，需要好好休息一下了。

总结

虽然这种效果现在来看并不惊艳了，但是其中的处理拉伸，剪切，弯曲的算法还是很值得一看。另一个值得一看是隐式求解步骤。

参考

[1]Baraff, D. and A. Witkin. “Large steps in cloth simulation.” *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998): n. pag.

[2]"Dynamic Deformables:Implementation and ProductionPracticalities "

[3]Tuur Stuyck Cloth Simulation for Computer Graphics 第七章

[4]https://github.com/davvm/clothSim

[5]R. Tamstorf, Derivation of discrete bending forces and their gradients 推导bend非常棒。